

MPFUN: A Portable High Performance Multiprecision Package

David H. Bailey

RNR Technical Report RNR-90-022

MPFUN: A Portable High Performance Multiprecision Package

David H. Bailey

December 18, 1990

Abstract

The author has written a package of Fortran routines that perform a variety of arithmetic operations and transcendental functions on floating point numbers of arbitrarily high precision, including large integers. This package features (1) virtually universal portability, (2) high performance, especially on vector supercomputers, (3) advanced algorithms, including FFT-based multiplication and quadratically convergent algorithms for π , \exp and \log , and (4) extensive self-checking and debug facilities that permit the package to be used as a rigorous system integrity test.

This paper describes the routines in the MPFUN package and includes discussion of the algorithms employed, the implementation techniques, performance results and some applications. Notable among the performance results is that the MPFUN package runs up to 13 times faster than another widely used package on a RISC workstation, and it runs up to 154 times faster than the other package on a Cray supercomputer.

The author is with the NAS Applied Research Office, NASA Ames Research Center, Moffett Field, CA 94035.

1. Applications of Multiprecision Computation

One question that is always raised in discussions of multiprecision computation is what applications justify such a facility. In fact, a number of applications, some of them entirely practical, have surfaced in recent years.

One important area of applications is in pure mathematics. While some still dispute whether a computer calculation can be the basis of a formal mathematical proof, certainly computations can be used to explore conjectures and reject those that are not sound. Such computations can thus save pure mathematicians a great deal of time by allowing them not to waste time searching for proofs of false notions. On the other hand, it must be acknowledged that if a conjecture is confirmed by computation to very high precision, then its validity is extremely likely. One can even ask which is more firmly established, a theorem whose lengthy proof has been read only by two or three people in the world, or a conjecture that has been independently confirmed by a number of high precision computations?

Some particularly nice applications of high precision computation to pure mathematics include the disproof of the Mertens conjecture by A. M. Odlyzko and H. J. J. te Riele [21], the disproof of the Bernstein conjecture in approximation theory by Varga and Carpenter [24] and the resolution of the “one-ninth” conjecture [24]. A number of other examples of multiprecision applications in analysis, approximation theory and numerical analysis are also described in [24].

One area in which multiprecision computations are especially useful is the study of mathematical constants. For example, although Euler’s constant γ is believed to be transcendental, it has not been proven that γ is even irrational. There is similar ignorance regarding other classical constants, such as $\log \pi$ and $e + \pi$, and also regarding some constants that have arisen in twentieth century mathematics, such as the Feigenbaum δ constant (4.669201609...) [14] and the Bernstein β constant (0.2801694990...) [24].

However, in most of these cases algorithms are known that permit these numbers to be computed to high precision. When this is done, the hypothesis of whether a constant α satisfies some reasonably simple, low-degree polynomial can be tested by computing the vector $(1, \alpha, \alpha^2, \dots, \alpha^{n-1})$ and then applying one of the recently discovered integer relation finding algorithms [4, 15, 17]. Such algorithms, when applied to an n -long vector x , determine whether there exist integers a_i such that $\sum a_i x_i = 0$. Thus if a computation finds such a set of integers a_i , these integers are the coefficients of a polynomial satisfied by α . Even if no such relation is found, these algorithms also produce bounds within which no relation can exist, which results are of interest by themselves. Clearly such analysis can be applied to any constant that can be computed to sufficiently high precision.

The author has performed some computations of this type [2, 4], and others are in progress. Some results that have been obtained include the following: (1) γ does not satisfy any polynomial of degree 12 or less with coefficients of Euclidean norm 1.04×10^{11} or less, (2) $\log \pi$ does not satisfy any polynomial of degree eight or less with coefficients of Euclidean norm 2.32×10^{37} or less, and (3) the imaginary part of the first zero of Riemann’s ζ function does not satisfy any polynomial of degree 12 or less with coefficients of Euclidean norm 4.71×10^{14} or less. In one case the author, working in conjunction with

H. R. P. Ferguson, obtained the following positive result: the third bifurcation point of the chaotic iteration $x_{k+1} = rx_k(1 - x_k)$, namely the constant $3.54409035955 \dots$, satisfies the polynomial $4913 + 2108t^2 - 604t^3 - 977t^4 + 8t^5 + 44t^6 + 392t^7 - 193t^8 - 40t^9 + 48t^{10} - 12t^{11} + t^{12}$.

One of the oldest applications of multiprecision computation is to explore the perplexing question of whether the decimal expansions (or the expansions in any other radix) of classical constants such as π , e , $\sqrt{2}$, etc. are random in some sense. Almost any reasonable notion of randomness could be used here, including the property that every digit occurs with limiting frequency $1/10$, or the stronger property that every n -long string of digits occurs with limiting frequency 10^{-n} . This conjecture is believed to hold for a very wide range of mathematical constants, including all irrational algebraic numbers and the transcendental π and e , among others. Its verification for a certain class of constants would potentially have the practical application of providing researchers with provably reliable pseudorandom number generators. Unfortunately, however, this conjecture has not been proven in even a single instance among the classical constants of mathematics. Thus there is continuing interest in computations of certain constants to very high precision, in order to see if there are any statistical anomalies in the results. The computation of π has been of particular interest in this regard, and recently the one billion digit mark was passed by both Kanada [18] and the Chudnovskys [12]. Statistical analyses of these results have so far not yielded any statistical anomalies (see for example [1]).

An eminently practical application of multiprecision computation is the emerging field of public-key cryptography, in particular research on the Rivest-Shamir-Adleman (RSA) cryptosystem [22, 13]. This cryptosystem relies on the exponentiation of a large integer to a large integer power modulo a third large integer. The RSA cryptosystem has also spawned a great deal of research into advanced algorithms for factoring large integers, since the RSA system can be “broken” if one can factor the modulus. The most impressive result in this area so far is the recent factorization of the ninth Fermat number $2^{512} + 1$, an integer with 155 digits, which was accomplished by means of numerous computer systems communicating by electronic mail. This computation employed a new factoring algorithm, known as the “number field sieve” [20].

An indirect application of multiprecision computation is the integrity testing of computer systems. A unique feature of multiprecision computations is that they are exceedingly unforgiving of hardware or compiler error. This is because if even a single computational error occurs, the result will almost certainly be completely incorrect after a possibly correct initial section. In many other scientific computations, a hardware error in particular might simply retard the convergence to the correct solution. On the other hand, if the result of a large multiprecision computation is correct, then the system has most likely performed millions or even billions of operations without error.

Not only do the final results constitute an integrity test, but there are several consistency checks that can be performed in the course of a multiprecision computation. One of these derives from the fact that when performing multiprecision multiplication using a complex fast Fourier transform (FFT), the final inverse FFT results should be quite close to integer values. If any result exceeds a nominal distance from an integer value, a hardware error

has likely occurred. Details are given in section 5.

The author's experience using current and prior versions of this package has confirmed this principle. Systems in which errors have been disclosed by the MPFUN routines include the following: the Cray-2 hardware, the Cray-2 CFT and CFT-77 Fortran compilers, the Cray X-MP CFT Fortran compiler, the ETA-10 hardware, the ETA-10 Fortran compiler, the DEC VAX 11/780 Fortran compiler, the Silicon Graphics IRIS hardware, the Silicon Graphics IRIS Fortran compiler, the Sun-4 Fortran compiler and the Intel iPSC-860 Fortran compiler. This list includes virtually all computer systems that the author has worked with in the last few years! Fortunately most of these problems have subsequently been rectified.

2. A Comparison of MPFUN with Other Multiprecision Systems

Several software packages are available for multiprecision computation. One that has been around for a while is the Brent MP multiprecision package, authored by R. P. Brent [10]. This package has the advantage of being freely available either from the author or from various other sources. It is very complete, including detailed numerical controls and many special functions.

Another package available at some sites is MACSYMA, which was originally developed at MIT but is now distributed by Symbolics, Inc. MACSYMA is actually a complete symbolic mathematics package, and its multiprecision arithmetic capability is only one part. A newer package of this sort is Mathematica, distributed by Wolfram Research, Inc. It features support of impressive full-color graphics for owners of advanced workstations.

There exist a number of other multiprecision systems in use that are specifically targeted for a particular computer system or for special applications. A package for large integer computation, with a focus on the Cray-2, is described in [11]. With the multiprecision computation tools currently available, some may question the need for yet another. In this regard, the author has attempted to combine some of the more valuable features of existing packages with a high performance design.

First of all, like Brent's MP package, the author's MPFUN package is freely available (within the USA), whereas the commercial products typically have hefty price tags and annual maintenance fees. Secondly, MPFUN runs virtually without change on any scientific computer, whereas most of the others require significant customization from system to system. As a result, an application written to call the author's routines on a workstation or even on a personal computer can be effortlessly ported to a more powerful system, such as a supercomputer, for extended computations.

One key feature of the MPFUN package is that it was written with a vector supercomputer or RISC floating point computer in mind from the beginning. Virtually all inner loops are vectorizable and employ floating point operations, which have the highest performance on supercomputers. As a result, MPFUN exhibits excellent performance on these systems. None of the other widely available packages, to the author's knowledge, exhibits respectable performance on supercomputers such as Crays. Also, the package avoids constructs that inhibit multiple processor computation. As a result, it can easily be modified to employ multitasking software.

Multiprecision numbers are represented in the MPFUN package as vectors of floating-point data with a dynamically variable length. Some other multiprecision systems, which feature a fixed precision level, often waste considerable time by performing operations on words containing zeroes. This dynamic precision level feature also means that MPFUN can be used efficiently for high-precision integer as well as non-integer calculations.

A final distinguishing feature of the MPFUN package is its usage of advanced algorithms. For many functions, both a "basic" and an "advanced" routine are provided. The advanced routines employ advanced algorithms and exhibit superior performance for extra-high precision (i.e. above about 1000 digit) calculations. For example, an advanced multiplication routine is available that employs a fast Fourier transform (FFT), and routines implementing the new Borwein quadratically convergent algorithms for exp and log are also provided.

3. Overview of the Package

The MPFUN package consists of approximately 8600 lines of Fortran code organized into 68 subprograms. These routines operate on two custom data types, multiprecision (MP) numbers and double precision plus exponent (DPE) numbers.

An MP number consists of a vector of single precision floating point numbers. The sign of the first word is the sign of the MP number. The magnitude of the first word is the count of mantissa words. The second word of the MP vector contains the exponent, which represents the power of 1,000,000. Words beginning with the third word in the array contain the mantissa. Mantissa words are floating point whole numbers between 0 and 999,999. For MP numbers with zero exponent, the "decimal" point is assumed after the first mantissa word. For example, the MP number 3 is represented by the three-long single precision vector (1., 0., 3.), and -123456789.012345 is represented by the five-long vector $(-3., 1., 123., 456789., 12345.)$. An MP zero is represented by the two-long vector (0., 0.).

If sufficient memory is available, the maximum precision level for MP numbers is approximately 16 million digits. The limiting factor for this precision level is the accuracy of calculations in the FFT-based multiplication routine. Beyond this level, rounding the double precision results of the final FFT to nearest integer is no longer reliable (see section 5 below). The maximum dynamic range of MP numbers is $10^{\pm 24,000,000}$ on 32 bit systems, and is higher still on 64 bit systems such as Crays.

A DPE number consists of a pair (A, N) , where A is a double precision scalar in the range $1 \leq |A| < 10$ and N is an integer. It represents $A \cdot 10^N$. DPE numbers are useful in multiple precision applications to represent numbers that do not require high precision but may have large exponent ranges. A DPE zero is denoted by the pair (0., 0.).

One may wonder why MP numbers are represented using floating point data, and why a decimal radix was chosen. The first decision derives from the fact that floating point performance is becoming the principal emphasis on almost all advanced scientific computers, from workstations to high-end supercomputers. This is particularly true on Cray systems, where the hardware instruction sets do not even include 64 bit integer multiplication or division instructions — such operations must be performed by first converting the argu-

ments to floating and then by using the floating point functional units. Basing MPFUN on floating point operations has the additional benefit that it permits virtually universal portability in the resulting program code. The selection of a decimal radix was based on the fact that on many systems floating point operations do not run any faster on power of two data than on other data, and a decimal radix is certainly preferable for debugging and maintenance.

There is one additional reason that the implementation is based on floating point arithmetic, and that the package may appear to be optimized for systems based on vector or RISC processors. This is because except for extremely high levels of precision (i.e. tens of thousands or millions of digits), there is not a great deal of low-level parallelism in multiprecision calculations. Thus except for modest-length vectors at the base level, multiprecision applications need to be parallelized at a higher level. For example, if one is performing computations with a matrix of multiprecision numbers, it is likely that parallelism can be exploited at the level of rows or columns of the matrix. This suggests that the preferred architecture for the parallel processing of multiprecision applications is a MIMD array of vector or RISC processors. Thus this code was thus written with such a computer model in mind.

MPFUN routines are available to perform the four basic arithmetic operations between MP numbers, to produce the integer and fractional parts, to produce a random MP number and to convert an MP number for input or output. Other routines perform operations between DPE numbers or between MP and DPE numbers, which saves time compared with performing these operations with the full MP routines. Some higher level routines compute complex products and quotients, square roots, cube roots, n -th powers, n -th roots, π , the functions exp, log, cos, sin, inverse cos and sin, the real or complex roots of polynomials and integer relations of real vectors. For many of these functions, both basic and advanced versions are available. The advanced routines employ advanced algorithms suitable for extra high precision computation.

Computations on large integers can be efficiently performed using this package by setting the working precision level two or three words higher than the largest integer that will be encountered (including products). These extra words of precision permit accurate integer division to be performed, using a multiprecision floating point division routine followed by a call to the routine that produces the integer and fractional parts of an MP number. There is no wasted computation when the actual size of an integer argument is much less than working precision level, since the MPFUN routines only perform arithmetic on the actual sizes of input data.

4. Portability and Testing

As mentioned earlier, one distinguishing feature of the MPFUN package is its portability. The standard version of MPFUN should run correctly, without alteration, on any computer with a Fortran-77 compiler that satisfies the following requirements [n and N denote integers]:

1. The truncation of a double precision value not exceeding 2^{30} , either by assignment to an integer variable or by using DINT, is correct.
2. The decimal-to-binary conversion of a double precision constant, which is either a whole number not exceeding 10^{14} in absolute value or the fraction $1/2$, is exact.
3. The decimal-to-binary conversion of any other double precision constant is correct to one part in 5×10^{13} .
4. The addition, subtraction, multiplication, truncated division, and exponentiation of integer variables or constants, where the arguments and results do not exceed 2^{30} in absolute value, produce exact results.
5. The addition, subtraction and multiplication of single precision variables or constants, where the arguments and results are whole numbers not exceeding 4×10^6 in absolute value, produce exact results.
6. The addition, subtraction and multiplication of double precision variables or constants, where the arguments and results are whole numbers not exceeding 10^{14} in absolute value, produce exact results.
7. The multiplication of a double precision variable with value 2^n , $-30 < n < 0$, by the fraction $1/2$ produces an exact result.
8. The addition, subtraction, multiplication and division of double precision variables or constants, where the arguments have values not mentioned above, produce results correct to within one part in 5×10^{13} .
9. The result of the operation $10.DO**N$, where N is between 0 and 12, is either exact or correct to within one part in 2×10^{13} . The results of $DLOG(X)$ and $DLOG10(X)$ for X between 10^{-12} and 10^{12} are correct to within one part in 2×10^{13} , except for X between 0.1 and 10, where the results are correct to within 10^{-13} . The results of $DCOS(X)$ and $DSIN(X)$ for X between $-\pi$ and π are correct to within 5×10^{-14} . The result of $DATAN2(X, Y)$ for X and Y on the unit circle is correct to within 10^{-13} .

The author is not aware of any serious scientific computer system in use today that fails to meet these requirements. Any system based on the IEEE 754 floating point standard, with a 25 bit mantissa in single precision and a 53 bit mantissa in double precision, easily meets these requirements. All DEC VAX systems meet these requirements. All IBM mainframes and workstations meet these requirements. Cray systems meet all of these requirements with double precision disabled (i.e. by using only single precision).

For IEEE systems or others with 52 or more bits in double precision data, an optional modification yields a two-fold performance improvement in the advanced multiplication routine for a certain range of precision levels. Details are given in the next section. There are a few other places in the program file where some simple modifications can be made

to optimize performance on a particular system. No more than ten lines of code need to be changed to tune the package for any system.

To insure that these routines are working correctly, a test suite is available. It exercises virtually all of the routines in the package and checks the results. This test program is useful in its own right as a computer system integrity test. As mentioned in the introduction, versions of this program have on numerous occasions disclosed hardware and software bugs in scientific computer systems.

5. The Four Basic Arithmetic Operations

Multiprecision addition and subtraction are not computationally expensive compared to multiplication, division, and square root extraction. Thus simple algorithms suffice to perform addition and subtraction. The only part of these operations that is not immediately conducive to vector processing is releasing the carries for the final result. This is because the normal “schoolboy” approach of beginning at the last cell and working forward is a recursive (i.e. non-vectorizable) operation. On a vector computer this is better done by starting at the beginning and releasing the carry only one cell back for each cell processed. Unfortunately, it cannot be guaranteed that one application of this process will release all carries. Thus it is necessary to repeat this operation until all carries have been released, usually only one or two additional times. In the rare cases where three applications of this operation are not successful in releasing all carries, the author’s program resorts to the scalar “schoolboy” method. On scalar or RISC computers, only the “schoolboy” scheme is used.

A key component of a high-performance multiprecision arithmetic system is the multiply operation, since in real applications typically a large fraction of the total time is spent here. The author’s basic multiply routine, which is used for modest levels of precision, employs a conventional “schoolboy” scheme, although care has been taken to insure that the operations are vectorizable. A significant saving is achieved by not releasing the carries after each vector multiply operation, but instead waiting until 64 such vector multiply operations have been completed. An additional saving is achieved by computing only the first half of the multiplication “pyramid”.

The schoolboy scheme for multiprecision multiplication has computational complexity proportional to n^2 , where n is the number of words or digits. For higher precision levels, other more sophisticated techniques have a significant advantage, with complexity as low as $n \log n \log \log n$. The history of the development of advanced multiprecision multiplication algorithms will not be reviewed here. The interested reader is referred to Knuth [19]. Because of the difficulty of implementing these advanced schemes and the widespread misconception that these algorithms are not suitable for “practical” application, they are rarely employed. For example, none of the widely used multiprecision packages employs an “advanced” multiplication algorithm. One instance where an advanced multiplication technique was employed is [13].

The author has implemented a number of these schemes, including variations of the Karatsuba-Winograd algorithm [19, p. 278] and schemes based on the discrete Fourier

transform (DFT) in various number fields [19, p. 290]. Based on performance studies of these implementations, the author has found that a scheme based on complex DFTs appears to be the most effective and efficient for modern scientific computer systems. The complex DFT and the inverse complex DFT of the sequence $x = (x_0, x_1, x_2, \dots, x_{N-1})$ are given by

$$F_k(x) = \sum_{j=0}^{N-1} x_j e^{-2\pi i j k / N}$$

$$F_k^{-1}(x) = \frac{1}{N} \sum_{j=0}^{N-1} x_j e^{2\pi i j k / N}$$

Let $C(x, y)$ denote the circular convolution of the sequences x and y :

$$C_k(x, y) = \sum_{j=0}^{N-1} x_j y_{k-j}$$

where the subscript $k - j$ is to be interpreted as $k - j + N$ if $k - j$ is negative. Then the convolution theorem for discrete sequences states that

$$F[C(x, y)] = F(x)F(y)$$

or expressed another way

$$C(x, y) = F^{-1}[F(x)F(y)]$$

This result is applicable to multiprecision multiplication in the following way. Let x and y be n -long representations of two multiprecision numbers (without the sign or exponent words). Extend x and y to length $2n$ by appending n zeroes at the end of each. Then the multiprecision product z of x and y , except for releasing the carries, can be written as follows:

$$\begin{aligned} z_0 &= x_0 y_0 \\ z_1 &= x_0 y_1 + x_1 y_0 \\ z_2 &= x_0 y_2 + x_1 y_1 + x_2 y_0 \\ &\vdots \\ z_{n-1} &= x_0 y_{n-1} + x_1 y_{n-2} + \dots + x_{n-1} y_0 \\ &\vdots \\ z_{2n-3} &= x_{n-1} y_{n-2} + x_{n-2} y_{n-1} \\ z_{2n-2} &= x_{n-1} y_{n-1} \\ z_{2n-1} &= 0 \end{aligned}$$

It can now be seen that this multiplication pyramid is precisely the convolution of the two sequences x and y , where $N = 2n$. In other words, the multiplication pyramid can be obtained by performing two forward DFTs, one vector complex multiplication, and one inverse DFT, each of length $N = 2n$. Once the inverse DFT results have been adjusted to the nearest integer to compensate for any numerical error, the final multiprecision product may be obtained by merely releasing the carries as described above.

The computational savings arises from the fact that complex DFTs may of course be economically computed using some variation of the fast Fourier transform (FFT) algorithm. The particular FFT algorithm utilized for the MPFUN advanced multiplication routine is described in [3]. It was first proposed by Swarztrauber and is sometimes called the "Stockham-Transpose-Stockham" FFT. This algorithm features reasonably high performance on most computers, including vector and cache memory systems, and it can easily be modified for multiple processor computation if desired. For the implementation in this package, different techniques are employed for the matrix transposition step depending on the computer system and the size of the array. Since in this application the two inputs and the final output of the convolution are purely real, an algorithm is employed that converts the problem of computing the FFT on real data to that of computing the FFT on complex data of half the size. This results in a computational savings of approximately 50 percent.

One important detail has been omitted from the above discussion. Since the radix of MP numbers is 10^6 , the products $x_j y_{k-j}$ are in the neighborhood of 10^{12} , and the sum of a large number of these products cannot be represented exactly as a 64 bit floating point value, not matter how it is computed. In particular, the nearest integer operation at the completion of the final inverse FFT cannot reliably recover the exact multiplication pyramid result. For this reason, in the standard version of MPFUN, six digit data is always split into two words of three digits each upon entry to the FFT-based multiply routine. This permits computations of up to approximately 16 million digits to be performed correctly.

Included in the advanced multiply routine (although normally commented out) is some code that determines the maximum FFT roundoff error, i.e. the maximum difference between the final FFT results and the nearest integer values, and tests if it is greater than a certain reliable level. This code can also be used as a system integrity test, since for modest levels of precision with splitting, the maximum FFT roundoff error should be a rather small number, and an excessive value indicates that a hardware or compiler error has occurred.

On systems based upon the IEEE 754 standard, with 53 mantissa bits in double precision, it is possible to avoid this splitting operation for precision levels up to about 6,000 digits. Thus up to this level of precision, the advanced multiply routine runs twice as fast, since FFTs of only half the normal size can be employed, although the FFT roundoff error must be checked on each result. This feature also permits the cross-over point for the advanced routines, i.e. the level of precision below which the advanced routines merely call the basic routines, to be set lower. This feature can be implemented in the program by changing a single line of code.

The division of two MP numbers of modest precision is performed using a fairly straight-

forward scheme. Trial quotients are computed in double precision, using up to three six digit words of the dividend and up to three words of the divisor. This guarantees that the trial quotient is virtually always correct. In those rare cases where one or more words of the quotient are incorrect, the result is automatically fixed in a cleanup routine at no extra computational cost.

In the advanced division routine, the quotient of a and b is computed as follows. First the following Newton-Raphson iteration is employed, which converges to $1/b$:

$$x_{k+1} = x_k(2 - bx_k)$$

Multiplying the final approximation to $1/b$ by a gives the quotient. Note that this algorithm involves only a simple subtraction, plus two multiplications, which can be performed using the FFT-based technique mentioned above.

Algorithms based on Newton iterations have the desirable property that they are inherently self-correcting. Thus these Newton iterations can be performed with a precision level that doubles with each iteration. One difficulty with this procedure is that errors can accumulate in the trailing mantissa words. This error can be economically controlled by repeating the next-to-last iteration. This increases the run time by only about 25 percent, and yet the result is accurate to all except possibly the last two words.

It can easily be seen that the total cost of computing a reciprocal by this means is about 2.5 times the cost of the final iteration. The total cost of a multiprecision division is only about six times the cost of a multiprecision multiplication operation of equivalent size.

6. Other Algebraic Operations

Complex multiprecision multiplication is performed using the identity

$$(a_1 + a_2i)(b_1 + b_2i) = [a_1b_1 - a_2b_2] + [(a_1 + a_2)(b_1 + b_2) - a_1b_1 - a_2b_2]i$$

Note that this formula can be implemented using only three multiprecision multiplications, whereas the straightforward formula requires four. Complex division is performed using the identity

$$\frac{a_1 + a_2i}{b_1 + b_2i} = \frac{(a_1 + a_2i)(b_1 - b_2i)}{b_1^2 + b_2^2}$$

where the complex product in the numerator is evaluated as above. Since division is significantly more expensive than multiplication, the two real divisions ordinarily required in this formula are replaced with a reciprocal computation of $b_1^2 + b_2^2$ followed by two multiplications. The advanced routines for complex multiplication and division utilize these same formulas, but they call the advanced routines for real multiplication and division.

The general scheme described in the last section to perform division by Newton iterations is also employed to evaluate a number of other algebraic operations. For example, square roots are computed by employing the following Newton iteration, which converges to $1/\sqrt{a}$:

$$x_{k+1} = \frac{x_k}{2}(3 - ax_k^2)$$

Multiplying the final approximation to $1/\sqrt{a}$ by a gives the square root. As with division, these iterations are performed with a precision level that approximately doubles with each iteration. The basic square root routine computes each iteration to one word more than a power of two. As a result, errors do not accumulate very much, and it suffices to repeat the third-from-the-last iteration to insure full accuracy in the final result. The added cost of repeating this iteration is negligible.

The advanced square root routine cannot compute each iteration to one greater than a power of two words, since the levels of precision are restricted to exact powers of two by the FFT-based multiply procedure. Thus the advanced routine repeats the next-to-last iteration. As in the advanced divide routine, repeating the next-to-last iteration adds about 25 percent to the run time.

Cube roots are analogously computed by the following Newton iteration, which converges to $a^{-2/3}$:

$$x_{k+1} = \frac{x_k}{3}(4 - a^2 x_k^3)$$

Multiplying the final approximation to $a^{-2/3}$ by a gives the cube root.

Included in the MPFUN package are a basic and an advanced routine to compute the n -th power of a multiprecision number. This operation is performed using the binary rule for exponentiation [19, p. 442]. When n is negative, the reciprocal is taken of the final result.

Along with the n -th power routines are two n -th root routines. For most inputs, these roots are computed using the following Newton iteration, which converges to $a^{-1/n}$:

$$x_{k+1} = \frac{x_k}{n}(n + 1 - ax_k^n)$$

The reciprocal of the final approximation to $a^{-1/n}$ is the n -th root. These iterations are performed with a dynamic precision level as before. When the argument a is very close to one and n is large, the n -th root is computed instead by the binomial expansion

$$(1 + a)^{1/n} = 1 + \frac{a}{n} + \frac{(1-n)a^2}{2! n^2} + \frac{(1-n)(1-2n)a^3}{3! n^3} + \dots$$

which is more economical in such cases. This feature results in significant time savings in the advanced routines for exp and log, which call this routine.

The MPFUN package includes four routines for computing roots of polynomials. There is a basic and an advanced routine for computing real roots of real polynomials and complex roots of complex polynomials. Let $P(x)$ be a polynomial and let $P'(x)$ be the derivative of $P(x)$. Let x_0 be a starting value that is close to the desired root. These routines then employ the following Newton iteration, which converges directly to the root:

$$x_{k+1} = x_k - P(x_k)/P'(x_k)$$

These iterations are computed with a dynamic precision level scheme similar to the routines described above.

One requirement for this method to work is that the desired root is not a repeated root. If one wishes to apply these routines to find a repeated root, it is first necessary to reduce the polynomial to one that has only simple roots. This can be done by performing the Euclidean algorithm in the ring of polynomials to determine the greatest common divisor $Q(x)$ of $P(x)$ and $P'(x)$. Then $R(x) = P(x)/Q(x)$ is a polynomial that has only simple roots.

In the introduction, the usage of integer relation finding algorithms was mentioned in exploring the transcendence of certain mathematical constants. The author has tested two recently discovered algorithms for this purpose, the “small integer relation algorithm” in [17], which will be termed the HJLS routine from the initials of the authors, and the “partial sum of squares” (PSOS) algorithm of H. R. P. Ferguson [4]. While each has its merits, the author has found that the HJLS routine is generally faster. Thus it has been implemented in MPFUN. For those readers interested in the PSOS algorithm, a routine implementing it is also available from the author.

Since both the HJLS and PSOS algorithms are quite complicated, neither will be presented here. Interested readers are referred to the respective papers.

7. Computing π

The computation of π to high precision has a long and colorful history. Interested readers are referred to [5] for discussion of the classical history of computing π . Recently a number of advanced algorithms have been discovered for the computation of π that feature very high rates of convergence [7, 8]. The first of these was discovered independently by Salamin [23] and Brent [9] and is referred to as either the Salamin-Brent algorithm or the Gauss-Legendre algorithm, since the mathematical basis of this algorithm has its roots in the nineteenth century. This algorithm exhibits quadratic convergence, i.e. each iteration approximately *doubles* the number of correct digits. Subsequently the Borweins have discovered a class of algorithms that exhibit m -th order convergence for any m [7, 8].

The author has tested a number of these algorithms. Surprisingly, although the Borwein algorithms exhibit higher rates of convergence, the overall run time is generally comparable to that of the Salamin-Brent algorithm. Since the Salamin-Brent algorithm is simpler, it was chosen for implementation in MPFUN. It may be stated as follows. Set $a_0 = 1$, $b_0 = 1/\sqrt{2}$, and $d_0 = \sqrt{2} - 1/2$. Then iterate the following operations beginning with $k = 1$:

$$\begin{aligned} a_k &= (a_{k-1} + b_{k-1})/2 \\ b_k &= \sqrt{a_{k-1}b_{k-1}} \\ d_k &= d_{k-1} - 2^k(a_k - b_k)^2 \end{aligned}$$

Then $p_k = (a_k + b_k)^2/d_k$ converges quadratically to π . Unfortunately this algorithm is not self-correcting like algorithms based on Newton iterations. Thus all iterations must be done with at least the precision level desired for the final result.

8. Transcendental Functions

The basic routine for exp employs a modification of the Taylor's series for e^t :

$$e^t = (1 + r + \frac{r^2}{2!} + \frac{r^3}{3!} + \frac{r^4}{4!} \dots)^8 10^n$$

where $r = t'/8$, $t' = t - n \log 10$ and where n is chosen to minimize the absolute value of t' . Reducing t in this manner and dividing by eight insures that $-0.14 < r \leq 0.14$, which significantly accelerates convergence in the above series.

The advanced routine for exp employs a quadratically convergent algorithm first outlined by the Borweins in [6]. Very small inputs t cause numerical difficulties in the Borwein algorithm, so the basic routine is called for these values. Larger inputs are reduced as above to within one half of $\log 10$ in absolute value.

The functions $P(s)$ and $Q(s)$ will now be defined. Set $x_0 = s$ and $y_0 = 16/(1 - s^2)$. Then iterate the following until convergence:

$$\begin{aligned} x_{k+1} &= \frac{2\sqrt{x_k}}{x_k + 1} \\ y_{k+1} &= y_k \left(\frac{x_k + 1}{2} \right)^{2^{1-k}} \end{aligned}$$

The extraction of 2^k -th roots in the last line is performed by the advanced n -th root routine. $P(s)$ is then defined as the limiting value of y_k . To define $Q(s)$, set $a_0 = 1$, $b_0 = s$, $a'_0 = 1$, and $b'_0 = \sqrt{1 - s^2}$. Then iterate the following until convergence:

$$\begin{aligned} a_{k+1} &= (a_k + b_k)/2 \\ b_{k+1} &= \sqrt{a_k b_k} \\ a'_{k+1} &= (a'_k + b'_k)/2 \\ b'_{k+1} &= \sqrt{a'_k b'_k} \end{aligned}$$

$Q(s)$ is defined as the ratio of the limits of a_k and a'_k . With $P(s)$ and $Q(s)$ defined, e^t may be evaluated by using Newton iterations with a dynamic precision level to solve the equation $Q(s) = |t|/\pi$ for s , and then evaluating $P(s)$. As an initial value for these Newton iterations, the author has found that a double precision value of $s = e^{1-\pi^2/(2t)}$ suffices for small positive t . If $t < 0$, the reciprocal is taken of the final result.

Since the usual Taylor series expansions of $\log x$ converge quite slowly, the basic routine for log merely solves the equation $x = e^t$ for t using the basic exp routine and Newton iterations with a dynamic precision level. The run time of the basic log routine is only about 2.5 times that of the exp routine. The advanced routine for log is quite similar to the advanced routine for exp, except that Newton iterations are used to solve the equation $P(s) = x$ for s , and then $\pi Q(s)$ is evaluated.

It might be mentioned that quadratically convergent algorithms for exp and log were first presented by Brent in [9]. Based on the author's comparisons, however, Brent's algorithms are not quite as fast as the Borweins'. For this reason the Borwein algorithms were selected for inclusion in this package.

The basic routine for sin and cos utilizes the Taylor's series for sin s :

$$\sin s = s - \frac{s^3}{3!} + \frac{s^5}{5!} - \frac{s^7}{7!} \dots$$

where $s = t - a\pi/2 - b\pi/16$ and the integers a and b are chosen to minimize the absolute value of s . We can then compute

$$\begin{aligned}\sin t &= \sin(s + a\pi/2 + b\pi/16) \\ \cos t &= \cos(s + a\pi/2 + b\pi/16)\end{aligned}$$

by applying elementary trigonometric identities for sums. The sin and cos of $b\pi/16$ are of the form $0.5\sqrt{2 \pm \sqrt{2 \pm \sqrt{2}}}$. Reducing t in this manner insures that $-\pi/32 < s \leq \pi/32$, which significantly accelerates convergence in the above series.

An advanced routine for sin and cos could be obtained by performing the Borwein algorithms with complex arithmetic instead of real arithmetic. Unfortunately, the resulting routine would not be competitive with the above Taylor's series scheme unless a precision level of over 10,000 digits were used. Thus the advanced routine for sin and cos utilizes the same algorithm as the basic routine.

The basic and advanced routines for inverse sin and cos merely solve the system of equations $[\cos t = x, \sin t = y]$ using Newton iterations with a dynamic precision level.

9. Accuracy of Results

Most of the basic routines are designed to produce results correct to the last word of working precision, as is the advanced multiplication routine. Basic routines not guaranteed to the last word include the transcendental functions, where the accuracy of the results is limited by the accuracy of the input values π and $\log 10$. For the advanced routines other than multiplication, the last two to four words are not reliable, depending on the routine.

The accuracy of results from the MPFUN routines can be controlled by setting a rounding mode parameter. Depending on the value of this parameter, results are either truncated at the last mantissa word of working precision, or else the last word is rounded up depending on contents of the first omitted word.

Whichever routines and rounding mode are used, it is not easy to determine ahead of time what level of precision is necessary to produce results accurate to a desired tolerance. Also, despite safeguards and testing, a package of this sort cannot be warranted to be free from bugs. Additionally, compiler and hardware errors do occur, and it is not certain that they will be detected by the package. Thus the following procedure is recommended to increase one's confidence in computed results:

1. Start with a working double precision program, and then check that the ported multiprecision code duplicates intermediate and final results to a reasonable accuracy.
2. Where possible, use the ported multiprecision code to compute special values that can be compared with other published high precision values.

3. Repeat the calculation with the rounding mode parameter changed, in order to test the sensitivity of the calculation to numerical error. Alternatively, repeat the calculation with the precision level set to a higher level.
4. Repeat the calculation on another computer system, in order to certify that no hardware or compiler error has occurred.

10. Using the MPFUN Package

Specific instructions for converting an application to perform multiprecision arithmetic using the MPFUN routines are given in the appendix. The appendix also contains a complete list of the MPFUN routines, together with calling sequences and other information.

It should be mentioned that the task of converting an application to reference the MPFUN routines will be greatly facilitated when the Fortran-90 language standard is adopted. The latest draft of this standard [16], which is considered to be very close to the final version, provides for “derived data types” and “defined operations”. These features would allow one to define a new data type, MULTIPRECISION, and specify that certain variables have this type. Then by extending the standard arithmetic operators using an interface module, the appropriate MPFUN routine would be referenced whenever a multiprecision variable occurs in an expression. Even the operator = could be redefined in this way, so that for example conversion between double precision and multiprecision could be performed by a simple assignment statement. Usage of such features will of course have to await the final adoption of this standard and its implementation on a number of scientific computer systems.

11. Performance

Table 1 gives some performance results of the MPFUN package. These results compare this package with the Brent MP package, perhaps the most widely used multiprecision computation package. The problem selected for comparison is the computation of π using the Salamin-Brent algorithm, since this computation exercises all of the basic and advanced arithmetic and square root routines. Timings in seconds are included for both a RISC workstation and a supercomputer. The RISC workstation is one processor of the 4D-320 model from Silicon Graphics, Inc. (SGI), which has a theoretical peak performance of 16 MFLOPS and a Linpack performance of 4.9 MFLOPS (double precision figures). The supercomputer is one processor of the Cray Y-MP, which has a theoretical peak performance of 330 MFLOPS and a Linpack performance of 90 MFLOPS. When these runs were made, the SGI system was running IRIX 3.3 system software, and the Cray was running UNICOS 6.0. A blank in the table indicates that the run would have taken an unreasonable amount of time and was not performed. The number of digits in the second column is equal to $6 \cdot 2^M$.

It can be seen from these results that the MPFUN package is uniformly faster than the MP package on both systems. On the SGI system, MPFUN is only about twice as fast as MP for lower precision levels, but once the level of precision rises above 1000 digits, MPFUN has a considerable advantage, due mainly to its FFT-based multiply routine. At

12,288 digits precision, the highest level at which both programs could be compared, the MPFUN package is 13 times faster. The relatively sharp jump in MPFUN timings on the SGI system between $M = 10$ and $M = 12$ is due in part to the fact that the splitting operation described in section 5 cannot be avoided for M greater than ten.

On the Cray Y-MP, the results are even more favorable — the MPFUN package is four times faster at the lowest precision and 154 times faster at 49,152 digits precision, the highest level at which both could be compared. The fact that this ratio is much higher than on the SGI system (and therefore much higher than can be accounted for based on operation counts alone) is chiefly due to a very high level of vectorization in the innermost loops of the MPFUN routines, as well as their reliance on floating point instead of integer operations. The MPFUN run times on the Y-MP do not exhibit any sharp jumps because the splitting operation is always performed whenever the advanced multiplication routine is invoked on this system. At the highest precision level listed, the Y-MP is running the author's code at 188 MFLOPS, or 57% of the one processor peak rate.

M	Digits	MPFUN Times		MP Times	
		SGI	Y-MP	SGI	Y-MP
4	96	0.03	0.0054	0.05	0.0199
5	192	0.08	0.0085	0.13	0.0386
6	384	0.24	0.0147	0.41	0.0865
7	768	0.70	0.0304	1.52	0.2368
8	1,536	1.59	0.0776	6.07	0.5860
9	3,072	4.03	0.1752	25.60	2.1154
10	6,144	9.01	0.4249	108.17	8.5274
11	12,288	34.76	0.8837	454.56	34.9628
12	24,576	151.84	1.8617		146.8758
13	49,152	448.65	4.0248		619.6510
14	98,304		8.6533		
15	196,608		19.0777		
16	393,216		41.4193		
17	786,432		94.8807		
18	1,572,864		196.6377		

Table 1: Performance Results

References

1. Bailey, D. H., "The Computation of π to 29,360,000 Decimal Digits Using Borweins' Quartically Convergent Algorithm", *Mathematics of Computation*, vol. 50 (Jan. 1988), p. 283 - 296.
2. Bailey, D. H., "Numerical Results on the Transcendence of Constants Involving π , e , and Euler's Constant", *Mathematics of Computation*, vol. 50 (Jan. 1988), p. 275 - 281.
3. Bailey, D. H., "A High Performance FFT Algorithm for Vector Supercomputers", *International Journal of Supercomputer Applications*, vol. 2 (Spring 1988), p. 82 - 87.
4. Bailey, D. H., and Ferguson, H. R. P., "Numerical Results on Relations Between Numerical Constants Using a New Algorithm", *Mathematics of Computation*, vol. 53 (October 1989), p. 649 - 656.
5. Beckmann, P., *A History of Pi*, Golem Press, Boulder CO, 1977.
6. Borwein, J. M., and Borwein, P. B., "The Arithmetic-Geometric Mean and Fast Computation of Elementary Functions", *SIAM Review* vol. 26 (1984), p. 351 - 365.
7. Borwein, J. M., and Borwein, P. B., *Pi and the AGM*, John Wiley, New York, 1987.
8. Borwein, J. M., Borwein, P. B., and Bailey, D. H., "Ramanujan, Modular Equations, and Approximations to Pi", *The American Mathematical Monthly*, vol. 96 (1989), p. 201 - 219.
9. Brent, R. P., "Fast Multiple-Precision Evaluation of Elementary Functions", *Journal of the ACM*, vol. 23 (1976), p. 242 - 251.
10. Brent, R. P., "A Fortran Multiple Precision Arithmetic Package", *ACM Transactions on Mathematical Software*, vol. 4 (1978), p. 57 - 70.
11. Buell, D., and Ward, R., "A Multiprecise Integer Arithmetic Package", *Journal of Supercomputing*, vol. 3 (1989), p. 89 - 107.
12. Chudnovsky, D. V. and Chudnovsky, G. V., "Computation and Arithmetic Nature of Classical Constants", *IBM Research Report*, IBM T. J. Watson Research Center, RC14950 (#66818), 1989.
13. Comba, P. G., "Exponentiation Cryptosystems on the IBM PC", *IBM Systems Journal*, vol. 29 (1990), p. 526 - 538.
14. Feigenbaum, M. J., "Quantitative Universality for a Class of Nonlinear Transformations", *Journal of Statistical Physics*, vol. 19 (1978), p. 25 - 52.

15. Ferguson, H. R. P., and Forcade, R. W., "Generalization of the Euclidean Algorithm for Real Numbers to All Dimensions Higher Than Two", *Bulletin of the American Mathematical Society*, vol. 1 (1979), p. 912 - 914.
16. *Fortran 90*, draft working document of X3J3, American National Standards Institute, New York, June 1990.
17. Hastad, J., Just, B., Lagarias, J. C., and Schnorr, C. P., "Polynomial Time Algorithms for Finding Integer Relations Among Real Numbers", *SIAM Journal on Computing*, vol. 18 (1988), p. 859 - 881.
18. Kanada, Y., personal communication, 1989.
19. Knuth, D. E., *The Art of Computer Programming*, Addison Wesley, Menlo Park, 1981.
20. Lenstra, A. K., Lenstra, H. W., Manasse, M. S., Pollard, J. M., "The Number Field Sieve", *1990 ACM Symposium on the Theory of Computing*, p. 564 - 572.
21. Odlyzko, A. M. and te Riele, H. J. J., "Disproof of the Mertens Conjecture", *J. Reine Angew. Mathematik*, vol. 357 (1985), p. 138 - 160.
22. Rivest, R. L., Shamir, A., and Adleman, L., "A Method for Obtaining Digital Signatures and Public-Key Cryptosystems", *Communications of the ACM*, vol. 21 (1978), p. 120 - 126.
23. Salamin, E., "Computation of π Using Arithmetic-Geometric Mean", *Mathematics of Computation*, vol. 30 (1976), p. 565 - 570.
24. Varga, R. S., *Scientific Computation on Mathematical Problems and Conjectures*, SIAM, Philadelphia, 1990.

Appendix: Usage Instructions

The MPFUN routines are listed, together with a brief functional description in Tables 2 and 3. Note that no routines are provided for absolute value or negation, since these operations may be performed by merely taking absolute value of or negating the first word of the single precision vector representing an MP number. Before calling any of these routines, some integer parameters in common block MPCOM1 should be set. In order, they are NW, IDB, LDB, IER, MCR, IRD, ICS, IHS, and IMS. They are defined as follows:

1. NW is the maximum number of mantissa words. This should be set by the user in the main calling program to $ND/6$, where ND is the desired maximum precision level in digits. Some routines modify this parameter but restore it prior to exiting. Default: 16 (i.e. 96 digits).
2. IDB is a debug flag and ordinarily should be set to zero. Setting IDB to an integer between 4 and 10 produces debug printouts in varying degrees of detail from the subroutines of this package. Values of IDB between 1 and 3 are available for use as debug flags in the user's calling program if desired. Default: 0.
3. LDB is the logical unit number for output of debug and error messages. Default: 6.
4. IER is an error flag and should initially be set to zero. It is set to nonzero values by the routines when an error condition is detected.
5. MCR is the "crossover" point for the advanced routines — if an advanced routine is called with a precision level NW that is $2^{**}MCR$ or less, the advanced routine merely calls the basic MP routine. Default: 8 (7 on IEEE systems).
6. IRD controls the rounding mode: when $IRD = 0$, the last word is truncated, i.e. the same compared with higher precision; when $IRD = 1$, the last mantissa word is rounded up if the first omitted word is 500,000 or more; when $IRD = 2$, the last mantissa word is rounded up if the first omitted word is nonzero. Default: 1.
7. ICS is the current single precision scratch space stack pointer and should initially be set to one.
8. IHS is the current "high water mark" of ICS and should be initially set to one.
9. IMS is the maximum single precision scratch space available. Default: 1024.

Common block MPCOM2 contains an integer array KER of length 72. This array controls the action taken when one of the MP routines detect an error condition, such as an attempted divide by zero. If the entry corresponding to a particular error number is zero, the error is ignored and execution resumes, although the routine that detects the condition is usually exited. If the entry is one, a message is output on unit LDB, and IER is set to the error number (an integer between 1 and 72), but execution resumes. If the entry is

two, a message is output and execution is terminated. All entries of the KER array are initialized to two in a block data subprogram. The user may change some of these entries by including the common MPCOM2 in the user's program.

An MP argument may not be used as both as an input and an output variable in a call to one of the MP routines. Output arrays for holding MP results generally require $NW+4$ single precision cells. An exception is MPDMC, where the output MP variable only requires seven cells. MPCOUT produces literal results, which require $6*NW+20$ cells of type CHARACTER*1. Many of these routines require either single precision or double precision scratch space or both. Single precision scratch space is contained in common block MPCOM3, while double precision scratch space is contained in common block MPCOM4. As a default, the package allocates 1024 cells in each of these two blocks.

The amount of single precision scratch space needed for an application varies widely depending on the routines used and the precision level NW. The maximum amount for each routine is given in Tables 2 and 3. The simplest way to determine the proper amount for a program is to run the code with an ample amount and then output the value of the parameter IHS in common MPCOM1 upon completion. If insufficient space has been allocated, an error message will be output (error code 5) and execution will be terminated. If this occurs, the user must allocate a larger single precision array (with at least the number of words indicated in the error message) and place it in common MPCOM3 in the user's main program. In addition, the parameter IMS in MPCOM1 must be set to this larger number.

There is no nesting of double precision scratch space, and so the amount required is simple to determine. Let NX be the largest precision level NW used in an application. Then at most $NX+7$ double precision cells are required for the basic MP routines, and at most $12*NX+6$ cells are required for the advanced routines. If more than the default 1024 cells are required, the user must allocate a larger array and place it in common MPCOM4 in the user's main program. Since it is straightforward to determine ahead of time the required level of double precision scratch space, the package does not perform automatic checking as it does for single precision scratch space.

If any of the advanced routines (i.e. those whose name ends in X) is called, the user must also allocate and initialize an array in common MPCOM5. The advanced routines should be called with a level of precision NW that is a power of two, so let $NX = 2**MX$ be the largest level that will be used in a program. Then the user must allocate at least $8*NX$ double precision cells in common block MPCOM5 and must call MPINIX with argument MX to initialize the array in MPCOM5. Again, since the amount of space is straightforward to determine, no automatic checking is performed.

If the user does allocate arrays in MPCOM3, MPCOM4 or MPCOM5 with different sizes than the default 1024 cells, some systems may flag a warning message when the user's program is compiled and linked with the precompiled MPFUN package, since this usage is technically not allowed under a strict reading of the Fortran-77 standard. The author is not aware of any system that flags a fatal error for such usage, provided that the common blocks in the user's main program are at least as large as in this package.

The following strategy is recommended for converting an ordinary single or double precision program to use the MPFUN package. First of all, make sure all MP variables are declared single precision. Include the common MPCOM1 in the main program and each subroutine. Make sure all MP variables used in subroutines, including necessary scratch arrays, are passed as subroutine arguments. Convert function subprograms to subroutines, where an additional argument returns the MP function value.

In the main program, define a parameter, say NX, to be the value desired for the precision level NW. Then declare every scalar MP variable to have dimension NX+4 and prepend NX+4 to the dimension of each MP array. In subroutines, declare every MP scalar to have dimension NW+4 and prepend NW+4 to the dimension of every MP array. Whenever an element of an MP array is passed as an argument to a subroutine, including one of the multiprecision routines, prepend 1 to the subscript. For example, the double precision program

```

      PROGRAM SAMPLE
      DOUBLE PRECISION A, B, C, DOT, T
      PARAMETER (N = 25)
      DIMENSION A(N), B(N)
C
      DO 100 I = 1, N
        T = I
        A(I) = SQRT (T)
        B(I) = 2.DO * A(I)
100  CONTINUE
C
      C = DOT (N, A, B)
      WRITE (6, '(1PD20.10)') C
      STOP
      END
C
      FUNCTION DOT (N, A, B)
      DOUBLE PRECISION A, B, DOT, S
      DIMENSION A(N), B(N)
C
      S = 0.DO
C
      DO 100 I = 1, N
        S = S + A(I) * B(I)
100  CONTINUE
C
      DOT = S
      RETURN
      END

```

is transformed into

```

PROGRAM SAMPLE
REAL A, B, C, S
DOUBLE PRECISION T
CHARACTER*1 CX
C
C Set the precision level to 120 digits. CX holds character data for
C the output of C. D is a double precision scratch array needed for
C MPMUL and MPSQRT. S is a scratch array large enough to hold three
C MP temporaries.
C
PARAMETER (N = 25, ND = 120, NX = ND / 6)
DIMENSION A(NX+4,N), B(NX+4,N), C(NX+4), CX(6*NX+20), S(3*NX+12)
COMMON /MPCOM1/ NW, IDB, LDB, IER, MCR, IRD, ICS, IHS, IMS
C
C Set the parameter NW. Default values suffice for other parameters
C in MPCOM1. The default scratch space in MPCOM3 and MPCOM4 is sufficient.
C
NW = NX
C
DO 100 I = 1, N
T = I
CALL MPDMC (T, 0, S)
CALL MPSQRT (S, A(1,I))
CALL MPMULD (A(1,I), 2.DO, 0, B(1,I))
100 CONTINUE
C
CALL DOT (N, A, B, C, S)
CALL MPCOUT (C, CX, NN)
WRITE (6, '(60A1)') (CX(I), I = 1, NN)
STOP
END
C
SUBROUTINE DOT (N, A, B, C, S)
REAL A, B, C, S
DIMENSION A(NW+4,N), B(NW+4,N), C(NW+4), S(3*NW+12)
COMMON /MPCOM1/ NW, IDB, LDB, IER, MCR, IRD, ICS, IHS, IMS
C
C K0, K1 and K2 are the starting positions of three separate NW+4 long
C sections of the scratch array S.
C
K0 = 1
K1 = NW + 5
K2 = 2 * NW + 9
CALL MPDMC (0.DO, 0, S(K0))
C

```

```

      DO 100 I = 1, N
        CALL MPMUL (A(1,I), B(1,I), S(K1))
        CALL MPADD (S(K0), S(K1), S(K2))
        CALL MPEQ (S(K2), S(K0))
100  CONTINUE
C
      CALL MPEQ (S(K0), C)
      RETURN
      END

```

This strategy has several advantages. First of all, it is straightforward. Secondly, the user is spared the necessity of manually computing indices or offsets in subscripts of multiprecision arrays. Finally, changing the precision level and changing the dimensions of the multiprecision arrays throughout the program can both be accomplished by merely altering a single **PARAMETER** statement in the main program.

Notes for Tables 2 and 3.

In Tables 2 and 3, the argument **PI** denotes an MP value of π , which must have been previously computed by calling either **MPPI** or **MPPIX**. The argument **AL10** denotes an MP value of $\log 10$, which must have been previously computed by calling either **MPL0G** or **MPL0GX**. Notation such as **(A, N)** in the third column denotes a DPE number, which has value $A * 10^{**N}$. In the scratch space column, **NW** is the precision level parameter in common **MPCOM1**. All other variables, unless indicated otherwise, denote MP numbers.

1. **MPANG** and **MPANGX** compute the MP angle **A** subtended by the MP pair **[X, Y]** considered as a point in the x, y plane. This is more useful than an **arctan** or **arcsin** routine, since it places the result correctly in the full circle, i.e. in the interval $(-\pi, \pi]$.
2. **MPCMUL**, **MPCMLX**, **MPCDIV**, and **MPCDVX** perform complex multiplication and complex division. **L** is the offset between the real and imaginary parts of **A**, **B** and **C**. **L** must be at least **NW+4**.
3. **MPCINP** converts the **CHARACTER*1** string **A** of length **N** to MP form in **B**, while **MPCOUT** converts the MP number **A** into the **CHARACTER*1** string **B** of length **N**. Strings input to **MPCINP** must be in the format $10^{sa} x tb.c$ where **a**, **b** and **c** are digit strings, **s** and **t** are - or blank, and **x** is either **x** or *****. Blanks may be embedded anywhere. The digit strings **a** and **b** are limited to nine digits and 80 total characters each, including blanks. The exponent portion (i.e. the portion up to and including **x**) may optionally be omitted.
4. **MPCPOL** and **MPCPLX** find a complex root of the **N**-th degree polynomial whose complex MP coefficients are in **A** by Newton-Raphson iterations, beginning at the complex DPE value $(X1(1), NX(1)) + i (X1(2), NX(2))$, and return the complex MP root in **X**. The **N+1** coefficients a_0, a_1, \dots, a_N are assumed to start in locations **A(1)**,

Routine Name	Calling Sequence	Functional Description	Single Prec. Scratch Space
DPADD	(A, NA, B, NB, C, NC)	$(C, NC) = (A, NA) + (B, NB)$	
DPDIV	(A, NA, B, NB, C, NC)	$(C, NC) = (A, NA) / (B, NB)$	
DPMUL	(A, NA, B, NB, C, NC)	$(C, NC) = (A, NA) * (B, NB)$	
DPPWR	(A, NA, B, NB, C, NC)	$(C, NC) = (A, NA) ** (B, NB)$	
DPSQRT	(A, NA, B, NB)	$(B, NB) = \text{Sqrt}(A, NA)$	
DPSUB	(A, NA, B, NB, C, NC)	$(C, NC) = (A, NA) - (B, NB)$	
MPADD	(A, B, C)	$C = A + B$	
MPANG	(X, Y, PI, A)	$A = \text{Ang}[X, Y]$. See note 1.	14*NW+81
MPANGX	(X, Y, PI, A)	$A = \text{Ang}[X, Y]$. See note 1.	19*NW+101
MPCBRT	(A, B)	$B = A ** (1/3)$	3*NW+15
MPCBRX	(A, B)	$B = A ** (1/3)$	4.5*NW+27
MPCDIV	(L, A, B, C)	$C = A / B$ complex. See note 2.	5*NW+20
MPCDVX	(L, A, B, C)	$C = A / B$ complex. See note 2.	7*NW+28
MPCINP	(A, N, B)	Converts for input. See 3.	NW+4
MPCMLX	(L, A, B, C)	$C = A * B$ complex. See note 2.	4*NW+16
MPCMUL	(L, A, B, C)	$C = A * B$ complex. See note 2.	4*NW+16
MPCOUT	(A, N, B)	Converts for output. See 3.	
MPCPLX	(N, LA, A, X1, NX, LX, X)	Finds complex roots. See 4.	17.5*NW+115
MPCPOL	(N, LA, A, X1, NX, LX, X)	Finds complex roots. See 4.	15*NW+75
MPCSSN	(A, PI, X, Y)	$X = \text{Cos}[A], Y = \text{Sin}[A]$	9*NW+47
MPCSSX	(A, PI, X, Y)	$X = \text{Cos}[A], Y = \text{Sin}[A]$	10*NW+46
MPDEB	(CS, A)	Outputs A preceded by the character string CS.	
MPDIV	(A, B, C)	$C = A / B$	
MPDIVD	(A, B, N, C)	$C = A / (B, N)$. See note 5.	NW+4
MPDIVX	(A, B, C)	$C = A / B$	2*NW+8
MPDMC	(A, N, B)	$B = (A, N)$. See note 5.	
MPEQ	(A, B)	$B = A$	
MPEXP	(A, AL10, B)	$B = \text{Exp}[A]$	5*NW+25
MPEXPX	(T, PI, AL10, Z)	$Z = \text{Exp}[T]$	25*NW+120
MPINFR	(A, B, C)	$B = \text{Int}[A], C = \text{Frac}[A]$	
MPINIX	(M)	Initializes for extra high precision. See 6.	
MPINRL	(N, LX, X, MN, MT, LR, R, IQ)	Finds integer relations. See 7.	N1*(NW+4)
MPINRX	(N, LX, X, MN, MT, LR, R, IQ)	Finds integer relations. See 7.	N2*(NW+4)
MPLOG	(A, AL10, B)	$B = \text{Log}[A]$	8*NW+48
MPLOGX	(Z, PI, AL10, T)	$T = \text{Log}[Z]$	25*NW+120
MPMDC	(A, B, N)	$(B, N) = A$	
MPMUL	(A, B, C)	$C = A * B$	
MPMULD	(A, B, N, C)	$C = A * (B, N)$. See note 5.	NW+4
MPMULX	(A, B, C)	$C = A * B$	
MPNINT	(A, B)	$B = \text{Nint}[A]$	NW+4

Table 2: List of Routines

Routine Name	Calling Sequence	Functional Description	Single Prec. Scratch Space
MPNPWR	(N, A, B)	$B = A ** N$	2*NW+10
MPNPWX	(N, A, B)	$B = A ** N$	2*NW+8
MPNRT	(N, A, B)	$B = A ** (1/N)$	6*NW+32
MPNRTX	(N, A, B)	$B = A ** (1/N)$	7*NW+48
MPPI	(PI)	PI = Pi	7*NW+37
MPPIX	(PI)	PI = Pi	8*NW+38
MPPOL	(N, L, A, X1, NX, X)	Finds real roots of polys. See 8.	5*NW+25
MPPOLX	(N, L, A, X1, NX, X)	Finds real roots of polys. See 8.	7.5*NW+45
MPRAND	(A)	A = Random number in [0, 1].	
MPSQRT	(A, B)	$B = \text{Sqrt } [A]$	2*NW+10
MPSQRX	(A, B)	$B = \text{Sqrt } [A]$	3*NW+18
MPSUB	(A, B, C)	$C = A - B$	

Table 3: List of Routines, Cont.

$A(2*LA+1)$, $A(4*LA+1)$, etc. LA is the offset between the real and the imaginary parts of each input coefficient. Typically $LA = NW+4$. LX, also an input parameter, is the offset between the real and the imaginary parts of the result to be stored in X. LX must be at least $NW+4$.

- Accurate results are not guaranteed if the absolute value of the double precision argument (i.e. A in MPDMC or B in MPMULD and MPDIVD) is outside the range $(10^{-6}, 10^{12})$ or has more than 12 significant digits. The integer exponent N may have any value.
- MPINIX must be called prior to using any advanced routine (i.e. any routine whose name ends in X). Calling MPINIX with argument M initializes for precision level $NW = 2**M$. The user must also allocate $2**(M+3)$ double precision cells in an array in common MPCOM5. Only one call to MPINIX is required in a program.
- MPINRL and MPINRX search for integer relations among the entries of the N-long MP vector X. An integer relation is an integer vector r such that $r_1x_1 + r_2x_2 + \dots + r_nx_n = 0$. The entries of x are assumed to start at $X(1)$, $X(LX+1)$, $X(2*LX+1)$, etc. Typically LX is set to $NW+4$. MN is the \log_{10} of the maximum Euclidean norm of an acceptable relation. When it has been determined that there are no relations with norm less than $10**MN$, processing is stopped. MT is the \log_{10} of the tolerance used for checking that a putative relation is a real one. This parameter is necessary because sometimes these routines require a higher level of working precision than the level used to generate the inputs. IQ is set to 1 if the routine succeeds in recovering a relation within the required bound. Otherwise IQ is set to 0. When a relation vector is recovered, it is placed in R, beginning at $R(1)$, $R(LR+1)$, $R(2*LR+1)$, etc., where LR, like LX, is an input parameter. LR should be at least $MN/6+3$. The parameters N1 and N2 in the scratch space column denote $4*N**2 + 5*N + 13$ and $4*N**2 + 5*N + 14$,

respectively. It is recommended that users set the debug parameter IDB in common MPCOM1 to 5 for the first few times in order to gain insight into the operation of these routines.

8. MPPOL and MPPOLX find a real root of the N -th degree polynomial whose MP coefficients are in A by Newton iterations, beginning at the DPE value (X1, NX) and return the MP root in X. The $N+1$ coefficients a_0, a_1, \dots, a_N are assumed to start in locations A(1), A(L+1), A(2*L+1), etc. Typically $L = NW+4$.